**Image Classifier: Recognizing Common Pests with Machine Learning.**

Kyle Brown, Miguel Garnica, Michell Valdivia, and Ky Phan

Advisor: Dr. Adriano Cavalcanti.

Central Washington University - Computer Science.

March 10th, 2021

# Acknowledgements.

We thank Xiaoping Wu and all contributors of the IP102 Insect Pest Recognition Dataset. The dataset used by this project helped to supply training data for our machine learning model.

We would also like to thank the creators and authors of [insectimages.org](https://www.insectimages.org/) and all of their contributors for additional pest images which helped supply additional training data.

# Abstract.

A significant amount of the global food supply is lost to insect pests annually. Any software that can accurately recognize these common pests may prove useful in limiting the loss of global food crops. Through the power of machine learning, the group managed to create a deep convolutional neural network (CNN) image classifier with TensorFlow, capable of recognizing some of these pests. The resulting model was later built into BuggyAI - a web application that allows the user to upload their own images, and then classify said images as either aphids (Aphidoidea), leafhoppers (Cicadellidae), spotted lanternflies (Lycorma Delicatula), and mirid bugs (Miridae). Trained with a dataset which contains over 20,000 images, the generated prediction model managed to produce results with validation accuracy over 70 percent. The final software looks promising.

**Keywords:** *Bug, Bugs, Classification, Convolutional Neural Network, Insect, Insects, Identification, Machine Learning, Model, Pest, Pests, Recognition.*

**Table of Contents.**

[Acknowledgements. 2](#_Toc66278382)

[Abstract. 3](#_Toc66278383)

[Glossary of Terms. 6](#_Toc66278384)

[**I.** **Introduction.** 7](#_Toc66278385)

[**II.** **Image Classifier.** 8](#_Toc66278386)

[**1.** **Getting Data.** 8](#_Toc66278387)

[**2.** **Why TensorFlow.** 8](#_Toc66278388)

[**3.** **Preprocessing Data.** 9](#_Toc66278389)

[**4.** **Convolutional Neural Network.** 11](#_Toc66278390)

[**5.** **Convolutional Layer.** 13](#_Toc66278391)

[**6.** **Maximum Pooling.** 14](#_Toc66278392)

[**7.** **Classification.** 15](#_Toc66278393)

[**8.** **Findings & Results.** 15](#_Toc66278394)

[**III.** **Backend & Server.** 21](#_Toc66278395)

[**1.** **Flask REST API.** 21](#_Toc66278396)

[**2.** **Server Architecture.** 22](#_Toc66278397)

[**3.** **Server Specifications.** 23](#_Toc66278398)

[**IV.** **Frontend.** 24](#_Toc66278399)

[**1.** **Styling.** 24](#_Toc66278400)

[**2.** **User Interface.** 24](#_Toc66278401)

[**3.** **Navigation.** 24](#_Toc66278402)

[**V.** **Security.** 25](#_Toc66278403)

[**1.** **Reverse Proxy Server.** 25](#_Toc66278404)

[**2.** **Brute Force Prevention.** 25](#_Toc66278405)

[**3.** **Secure Domain.** 25](#_Toc66278406)

[**4.** **Features Not Needed.** 26](#_Toc66278407)

[**VI.** **Testing.** 27](#_Toc66278408)

[**1.** **Front End Testing.** 27](#_Toc66278409)

[**2.** **Back End Unit Testing.** 27](#_Toc66278410)

[**3.** **Server Testing.** 28](#_Toc66278411)

[**VII.** **Source Control.** 29](#_Toc66278412)

[**1.** **GitHub.** 29](#_Toc66278413)

[**VIII.** **Appendix.** 30](#_Toc66278414)

[**1.** **Manual.** 30](#_Toc66278415)

[**IX.** **References.** 36](#_Toc66278416)

**Table of Figures.**

[Figure 1: The proper input shape for a Conv2d layer. 9](#_Toc66278479)

[Figure 2: Simple convolutional neural network. 11](#_Toc66278480)

[Figure 3: The constructed deep convolutional neural network. 12](#_Toc66278481)

[Figure 4: Feature extraction. 13](#_Toc66278482)

[Figure 5: Maximum Pooling. 14](#_Toc66278483)

[Figure 6: A random picture of a red spider mite. 16](#_Toc66278484)

[Figure 7: Result from the final prediction model. 18](#_Toc66278485)

[Figure 8: The testing result of the model. 19](#_Toc66278486)

[Figure 9: Flask REST API Diagram. 22](#_Toc66278487)

[Figure 10: Client Server Flow. 23](#_Toc66278488)

# Glossary of Terms.

* HTML - HyperText Markup Language.
* CSS - Cascading Style Sheets.
* Bootstrap - An open source CSS framework.
* JSON - JavaScript Object Notation (A data structure commonly used for sending and receiving data).
* HTTP - HyperText Transfer Protocol (protocol for transferring HTML and data).
* POST - An HTTP request method to send data to a server.
* GET - An HTTP request method to retrieve data from a server
* Supervisor - A Unix based process control system to automate and control processes.
* SSH - Secure Shell is a cryptographic network protocol to ensure secure server connection.
* WSGI - Web Server Gateway Interface (calling convention for web servers to handle requests for Python applications).
* Gunicorn - A Python WSGI HTTP Server for UNIX.
* NGINX - A server used as a reverse proxy for the Gunicorn server.
* Flask - A micro web framework written in Python.
* API - Application Programming Interface (interface defining software interactions).
* REST API - Representational State Transfer (a specific type of API commonly used for HTTP protocols).
* BLOB - Binary Large Object (a form of storing files as binary usually in a relational database).
* CNN - Convolutional Neural Network.
* ML - Machine learning.

# 

# 

# **Introduction.**

According to the United Nations (UN), the global food production is enough to feed the entire world, as stated in one article: “The fact is that we are already producing more than enough food for everyone. Between 1960 and 2015, agricultural production tripled in size, growing much faster than the global population” [[1]](#_wtkxs95w77t8). Unfortunately, not all the food produced reaches the consumers, which is why world hunger still exists today. There are many reasons for this food loss. The common pests is one such reason. According to the Food and Agriculture Organization of the United Nation (FAO) in 2019, an estimated “20 to 40 percent of global crop production are lost to pests” annually [[2]](#_wtkxs95w77t8).

On the other hand, unrelated to the previous world hunger problem mentioned, machine learning has become more and more popular in the Computer Science community, thanks to the wide range of utility it can provide. Numerous businesses are looking for potential employees who are well-educated in the subject mentioned. Almost without doubt, being knowledgeable in machine learning will provide one with more lucrative career options.

Given this opportunity to both help mitigate the common pests problem, and to learn more about machine learning at the same time, the group decided to create a small program that can recognize some common pests using machine learning, as their capstone project. This paper was written to serve as a report detailing the group’s journey to develop BuggyAI - a web application that recognizes common pests using machine learning.

# 

# **Image Classifier.**

## **Getting Data.**

The first important step in developing BuggyAI, is to make an image classifier that can distinguish, at the very least, some species of pests. To accomplish this goal the group needs to gather a large image dataset (the bigger it is, the better). These images will be used to train the model, in a process called supervised training. Essentially, the images were separated into different categories before the training process. The machine learning algorithm should go through all the images, “learn” which image belongs to which class of pest, and use the information it has “learnt” to classify new images. The bigger the dataset, the more examples are available to learn, the better the image classifier will turn out to be.

The group was lucky enough to contact the researcher Xiaoping Wu, and asked the researcher for access to the IP102 dataset, which contains about 75,000 thousand images in total, belonging to 102 species of common pest in China.

The IP102 dataset itself can be acquired here: <https://github.com/xpwu95/IP102>.

In the original IP102 dataset, the images of the 102 species of common pest were further divided into images used for training, images used for validating, and images used for testing. The group simply split the images according to species. The reason for this will be explained later in the **Preprocessing Data** part of the paper.

Of course, this was not the final dataset the group used. As more progress was made, the original image dataset would be modified further to satisfy the group’s needs.

## **Why TensorFlow.**

After acquiring a large dataset, the group then moved on to pick a tool to build the image classifier for BuggyAI. The group decided that the best tool for the job would be TensorFlow - a popular open-source machine learning library, maintained by a big community. The large user base means if the group ever encounters a problem when using TensorFlow, the solution to the problem is more likely to be found. Afterall, more users mean more people who can potentially help the group. For some group members, using TensorFlow to build a model is easy and intuitive. Also, as of TensorFlow 2.0, Keras (an open-source software library that provides a Python interface for artificial neural networks) is now also included in TensorFlow as well. Since every group members know basic coding with Python, it became apparent that TensorFlow 2.0 will be the group’s choice for anything machine learning related.

## **Preprocessing Data.**

After deciding on the proper tool to use, the group then had to decide what type of model to use. A quick search in the Internet showed numerous sites recommending the Convolutional Neural Network (CNN) as the “go to” solution when building an image classifier. Hence, the group agreed to make the image classifier by constructing a Convolutional Neural Network using the Python programming language. The group wanted to use Python because all members had some level of experience with it.

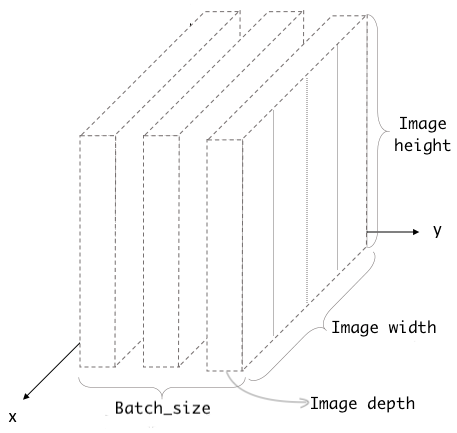


Figure 1: The proper input shape for a Conv2d layer.

Source: [[3]](#_wtkxs95w77t8).

The images had to be pre-processed before they were “fed” to the CNN as inputs. The previous figure 1 (on the previous page (pg. 8)) shows the proper input format for the CNN. The “Image height” and “Image width” limits how many pixels will be used (if the image is too large, it will be center cropped to the proper size). “Image depth” refers to the number of channels, which is usually three (red, green, blue, in that order). The paper will often refer to this as RGB channel for the rest of the paper. “Batch size” refers to the number of images being “fed” to the CNN each time. The final input shape for the group’s CNN looks like this:

* Image height = 180
* Image width = 180
* Image depth = 3
* Batch size = 32

As mentioned in the **Getting Data** part, in the original IP102 dataset, the images of the 102 species of common pest were further divided into images used for training, images used for validating, and images used for testing. The group simply split the images according to species. Only then would the images be split randomly into 80% for training, and 20% for validating. By randomly choosing which image would be used for training and which image would be used for validating, the group hoped the resulting model would be less biased.

To make the dataset more diverse, the group applied “data augmentation”, where the images were zoomed in, zoomed out, rotated (by various degrees, in various directions).

Finally, the values of the “Image depth” (the RGB channel) was scaled from [0,255] down to [0,1].

Also, the original IP102 dataset contains the images of the 102 species of common pests in China, many of which are rarely seen in the United States. Furthermore, since using all 75,000 images to train a model would take a considerable amount of time, the group initially decided to use only 3 pests: aphids, red spider mites, and thrips. After some experimentation, the group would switch to aphids (aphidoidea), leafhopper (cicadellidae), spotted lanternfly (lycorma delicatula), leaf bug (miridae), and a “none” class specifically images that did not belong to any of the 4 pest classes. More details regarding this change will be discussed later in the **Findings & Results** part of the paper.

## **Convolutional Neural Network.**

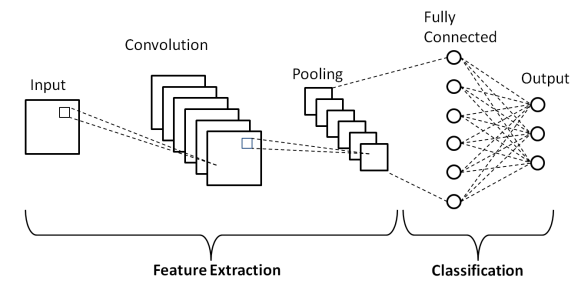


Figure 2: Simple convolutional neural network.

Source: [[4]](#_wtkxs95w77t8).

Figure 2 shows a simplified version of how a simple convolutional neural network can classify images. For the group, the inputs were the images previously processed. A batch of 32 images (the input), would go through the feature extraction process (going through a convolutional layer, and then a max pooling layer). After the feature extraction process, and then classified to (hopefully) appropriate output. Then the whole process (from input to output) was repeated using a new batch of 32 images, until all images in the dataset were used up. Again, figure 2 only shows a simplified version of how a simple convolutional neural network can classify images. The actual model the group built involved many more layers, as illustrated in figure 3 on the next page (pg. 11).

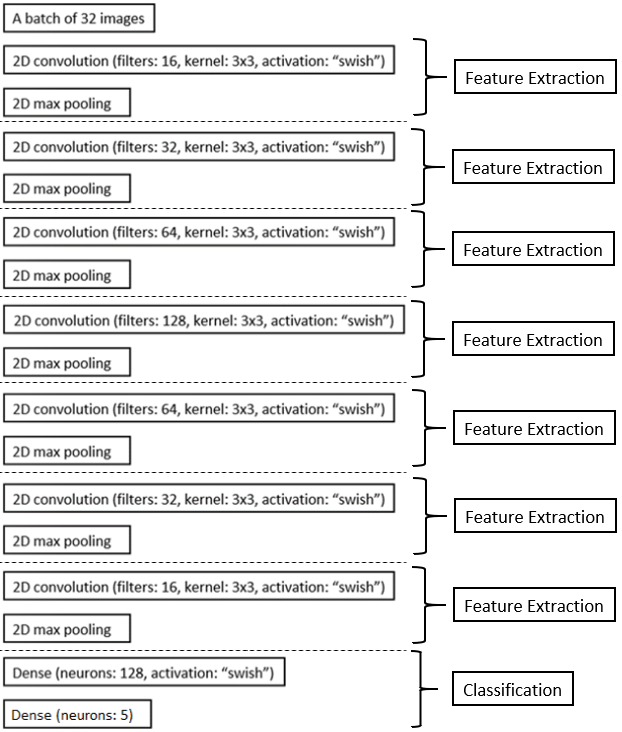


Figure 3: The constructed deep convolutional neural network.

Shown above, Figure 3, is the final structure of the deep convolutional neural network used to generate the prediction model. Before reaching the final layer, the inputs had to go through 5 feature extraction processes. Details on the feature extraction process will be discussed further later, in the **Convolutional Layer** and **Max Pooling** part. After the fifth feature extraction process, the resulting image tensor will be flattened into just a single array, which will be “fed” to a dense layer with 128 neurons, at the start of the classification process. The final layer is a dense layer with 5 neurons - one for each class: aphids (aphidoidea), leafhopper (cicadellidae), spotted lanternfly (lycorma delicatula), leaf bug (miridae), and a “none” class for images that did not belong to any of the 4 pest classes. The weight value of each of the 5 neurons in the final layer, ranges [0,1], represents the likelihood of the input image belonging to that class. Hence, getting the final output of the program is simply just returning which neuron has the highest weight value.

## **Convolutional Layer.**

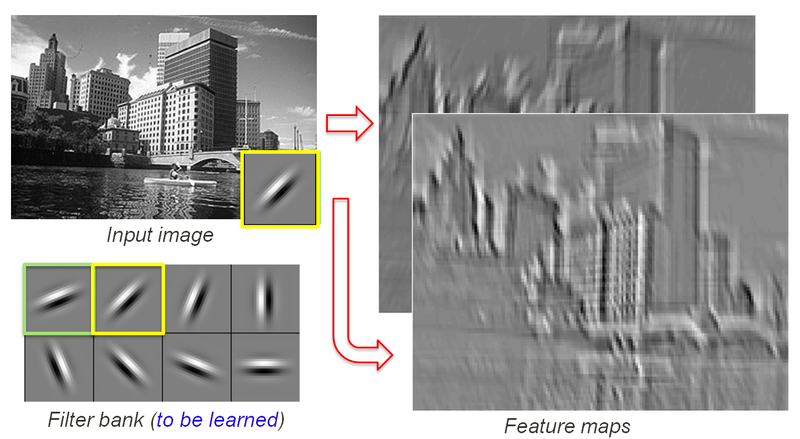


Figure 4: Feature extraction.

Source [[5]](#_wtkxs95w77t8).

As the name implies, convolutional layers are the building blocks of a convolutional neural network. Convolution is the process of applying a filter to an input that results in an activation whenever this filter pattern appears in the input picture. This filter is applied to an input image several times to generate a feature map, which is a map of activation.. The feature map indicates the locations and intensity of a feature detected in the image. This is how the model “see” the input image. Convolutional layers give neural networks the ability automatically to learn many features.

## **Maximum Pooling.**

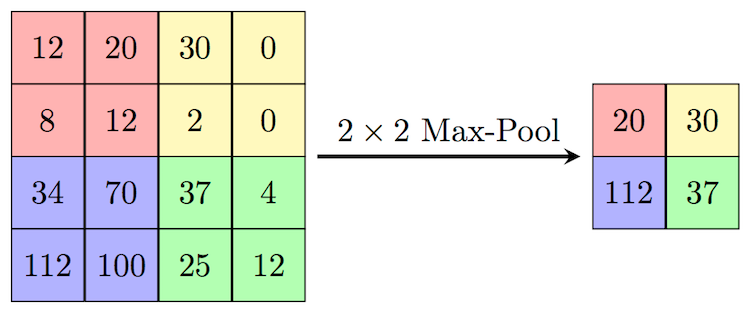


Figure 5: Maximum Pooling.

Source [[6]](#_wtkxs95w77t8).

A common issue with the output from feature maps is the sensitivity to the location of certain features in the image. A way to combat this is to down sample the output of feature maps. Maximum pooling or “max pooling” allows the feature maps to be more flexible to changes of the specific location of a feature. Max pooling and pooling layers summarize the features’ presence in a patch on the feature map. Show in figure 5, is an example of 2x2 max pooling, the maximum value from each 2x2 matrix is taken and the input shape is reduced. This takes the most activated feature or the highest value in the 2x2 matrix and reduces it to a singular grid space, thus preserving the presence of the feature while still down sampling the output.

In summary, the convolutional layers collect features from an image. The max pooling layer determines which features are activated most. Together, this pair of convolutional layer and max pooling layer forms the feature extraction process for the convolutional neural network.

## **Classification.**

After going through the 5 feature extraction processes, the image tensors were flattened. The flattening function would convert all the feature maps into a single one dimensional array for the next layers input. This sets up the input for the classification process that will identify the features and classify the image. The image tensors would then pass through a fully-connected layer with 128 neurons, allowing the network to summarize everything it had learnt from the images so far. The only thing left was to calculate the input images’s likelihood of belonging to each of the 5 classes. Hence, the final layer, also known as the output layer, is a dense layer with 5 neurons (the neurons representing the classes). After that, the software would just print out the class that has the highest likelihood.

After the CNN was completed, a prediction model was generated, and was saved as "buggyAI\_V4.1.h5" (without the quotation marks). This is the group’s prediction model.

## **Findings & Results.**

The model was modified numerous times throughout the development process.

As mentioned in the **Preprocessing Data** part of the paper, the group initially decided to use 3 pest classes: “aphids”, “red spider mites”, and “thrips”. However, the group encountered some major problems. The group noticed something when testing the first few versions of the model. Anytime the group fed images that were neither “aphids”, “red spider mites”, nor “thrips” to the model, it would classify them all as “aphids” with a level of confidence of around 75%. This error was later determined to be related to the available dataset.

* The “aphids” class contained around: 4100 images.
* The “red spider mites” class contained around: 500 images.
* The “thrips” class contained around: 900 images.

The vast majority of the images in the dataset, about 75%, that were used to train the model belonged to the “aphids” class. This matched the resulting level of confidence when the model was trying to classify images that were neither “aphids”, “red spider mites”, nor “thrips”.

The group made a conclusion. What the model had learnt was, if an image was given as input, the model could just classify the image as “aphids’ regardless of what the image looked like, and would still be right 75% of the time.

The group later changed the initial pest classes (aphids, red spider mites, thrips) to aphids (Aphidoidea), leafhoppers (Cicadellidae), spotted lanternflies (Lycorma Delicatula), and leaf bugs (Miridae). These classes have large sample sizes, each having over 4,000 images in the IP102 dataset. To increase the quality of the model, the group used Google and another website <https://www.insectimages.org/> to collect more images of the 4 pest classes, and to collect images for a “none” class meant for images that did not belong to any of the 4 pest classes. The group ended up with this final dataset:

* The Aphidoidea (aphids) class had around: 5900 images.
* The Cicadellidae (leafhopper) class had around: 6500 images.
* The Lycorma Delicatula (spotted lanternfly) class had around: 5300 images.
* The Miridae (leaf bug) class had around: 3600 images.
* The “none” class had around: 5800 images.

The dataset became “fairer”, with no pest class containing the vast majority of the images used to train the model. The result was an improvement.



Figure 6: A random picture of a red spider mite.

Most of the time, the model properly classified the pest. Even when the model made a wrong prediction, it did so with a much lower level of confidence, as it should. For example, when trying to classify an image of a random red spider mite (see figure 6 on previous page (pg. 15)), the model output was “This image most likely belongs to Miridae with a 41.29 percent confidence”. In short, the presence of the “none” class did reduce the model bias towards classes with large sample sizes. Further testing confirmed this was the case.

The group tried to improve the prediction model even more, through the process of trial and error (using different batch sizes, changing the number of filters in the convolutional layer, adding more dense layers with various neurons, etc.) before settling on the final configuration (see figure 3 again (pg. 11)). The performance of the latest prediction model is illustrated in figure 7 on the next page (pg. 17).

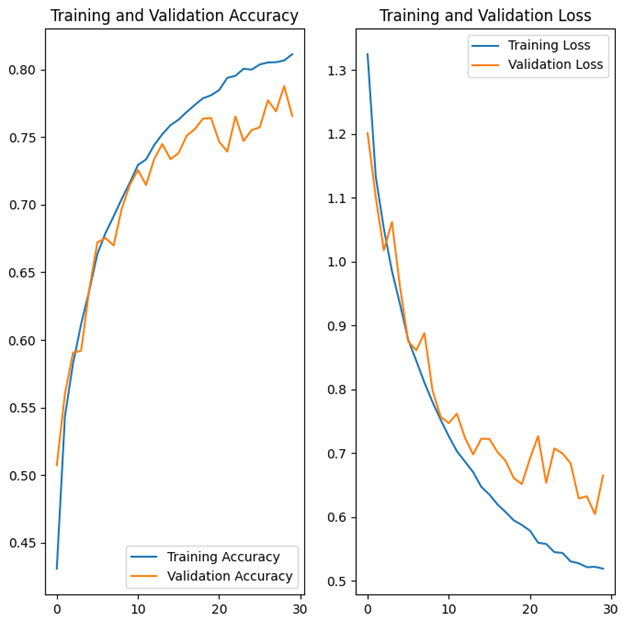


Figure 7: Result from the final prediction model.

The latest version of the image classifier reached almost 75% validation accuracy after 20 training epochs. The group decided to cap the number of training epochs to 30, since any more than that would cause overfitting. Also, due to the large number of images in the final dataset, generating a model took a lot of time, requiring about half an hour to generate just one model. Also, as seen in figure 7, the improvement rate of the validation accuracy is almost none after 30 epochs, signifying that the validation accuracy is nearly capped. The group concluded it was not worth it to train more than 30 epochs.



Figure 8: The testing result of the model.

As illustrated in figure 8, the performance of the model was satisfactory. It correctly classified most images. Some of the model’s drawbacks can be seen in figure 8 as well:

The image of the insect has to be detailed. This requires a close-up shot of the insect, which is unrealistic, as insects are easily scared. They will rarely stay still and wait for the photographer to capture a good picture of them. The model wrongly classified the image on the third row, third column as a spotted lanternfly (lycorma delicatula) since different insects look similar when seen from afar. Also, the pests often clump up, forming a colony. The image classifier has a difficult time trying to identify images of such colonies.

Green leafy background interference is also an issue. The pests love eating green leaves. They are almost always on a leaf. Hence, most training images of the pests also have a green leafy background. This caused the software to wrongly classify an image simply because the input has a similar background pattern to some images in the dataset. With enough sample size, the model should learn that the background is inconsequential in predicting pest species. However, time was running out and the group still had much left to do. Everyone was quite satisfied with the resulting prediction model despite the imperfections.

Next the paper will discuss the **Backend & Server** side of the BuggyAI.

# **Backend & Server.**

## **Flask REST API.**

The group’s application runs on a REST API in the form of a Flask application which in production is run as a Gunicorn WSGI server. As mentioned in the section prior our model training and testing was developed in a local setting, and once the model is finished it is deployed onto our server. The model which has been saved as a “.h5” file extension is loaded once upon startup of the Flask application and lies dormant until needing to be used by incoming POST requests.

The API is fairly simple as the scope of our project does not require complex functionality. Currently there is one POST request handler which only accepts incoming images from the front end or other exterior domains. Besides this there are GET request handlers which correspond to every applicable web page.

The POST request handler receives a file object, which on the client side is checked first to be of the proper file type (.jpg/.jpeg), and conforming to other valid filename properties. Once the image is verified on the client side to be a valid file it is then sent to our server. Upon receiving the file, our POST request handler function will resize the image to the appropriate dimensions needed for Tensorflow/Keras functions used. After converting the image to the expected input dimensions the image is passed to a Tensorflow/Keras prediction function. The result from the prediction is saved as a softmax score which is then used in forming a classification response string. The HTTP response is returned back to the calling service as a JSON response.

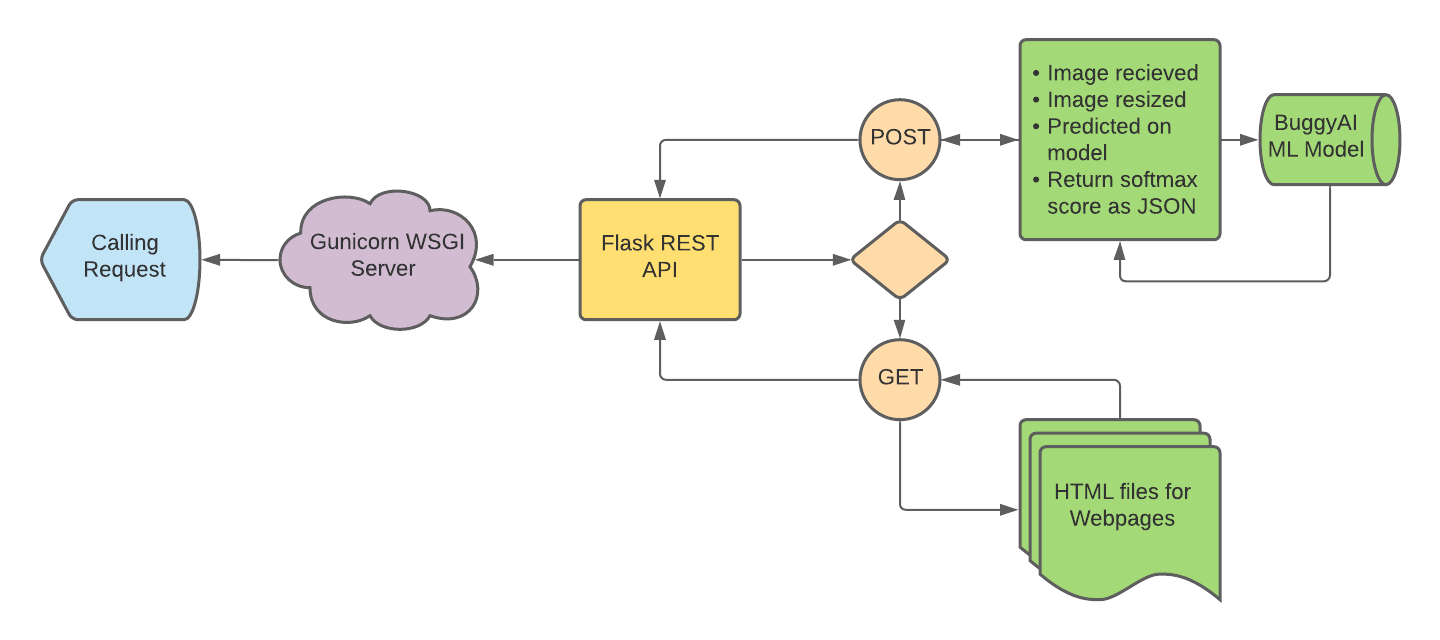


Figure 9: Flask REST API Diagram.

### 

## **Server Architecture.**

The BuggyAI application as a whole relies on the Flask REST API which resides on our Ubuntu 18.04 Linux server. At the entry point of our server we implement an Nginx server as a reverse proxy which routes incoming and outgoing requests as well as masking the web server’s identity. Upon receiving a valid request according to Nginx, the reverse proxy server will forward the request to our Gunicorn WSGI server.

The Gunicorn server is what is running the Flask application in a production setting and utilizes multiple worker processes to improve stability with more client requests per second. For a production or professional implementation, we use Supervisor which allows for the automation of processes such as the Nginx and Gunicorn Server startups upon restarting the physical Ubuntu server.

Upon receiving the incoming request from Nginx, the Gunicorn server will reference the Flask application which then executes the POST request handling function and returns a result. The result which is in the form of a JSON response will be passed back to the Nginx server. Nginx will then respond to the appropriate calling service and pass back the JSON response. Finally on the client side the user will see a response with a classification result populate in a small result window or an error message if something went wrong trying to classify the image. Pictured below is a server and API process flow from client to server ends.

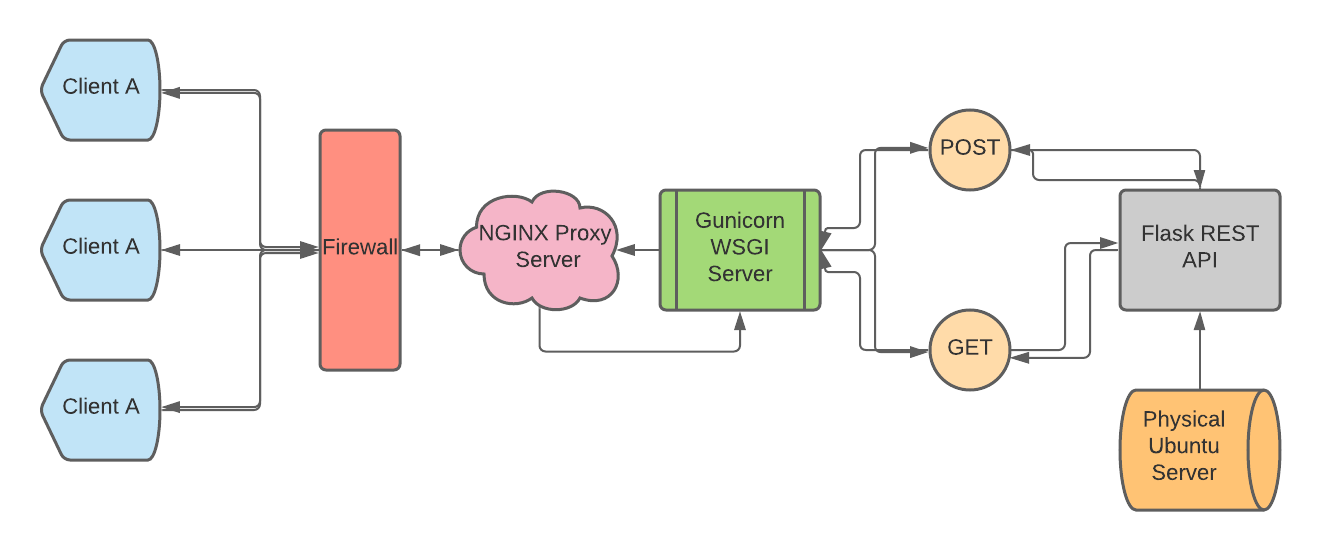


Figure 10: Client Server Flow.

### 

## **Server Specifications.**

As mentioned, the physical server we used is an Ubuntu 18.04 server which is a Debian based Linux distribution. Any other Ubuntu version or other Debian based Linux server should have the same behavior and allow other developers to implement what we have for our project.

In order to run a Tensorflow/Keras model comfortably with overhead CPU and RAM memory we chose to pay for a server which is affordable and allows 1 CPU core, 50GB of storage (for future use with implementing data collection), and 2GB of RAM. These server specs allow us to run 3 worker processes for our Gunicorn server meaning that there are 3 instasiations of the Flask application which serve requests and provide better performance than one instance. For more optimization and lateral scalability, we would be able to implement threaded workers which allow a combination of benefits from worker processes and threads. With our current configuration our Flask app is able to serve hundreds to a couple thousand requests per second.

# **Frontend.**

One can interact with the BuggyAi project by going to <https://www.buggyai.com/>. Through the website, the user can submit an image and request a classification on that image. The website uses HTML/CSS to style the frontend. The page navigation and display is handled by the Flask frontend python script.

The website consists of three pages: Home, About, and Manual. While these pages are distinct in their HTML code, they share the same stylesheet to maintain consistency.

## **Styling.**

The styling of the pages uses Bootstrap as its main styler. By using Bootstrap, the website will render consistently on multiple platforms, including mobile devices. Small changes were applied to the general styling such as colors, heights, and width adjustments. Colors were chosen using a palette generator site, Coolors.co.

The font for the header and page title use open source Google Fonts.

## **User Interface.**

The Homepage of BuggyAI.com is where most user interaction takes place. In this page the user can select an image from their current device, upload it, and receive a classification. The classification is displayed on a text box underneath the user’s image.

## **Navigation.**

To navigate between pages, the user can use the navigation bar located at the top of the page. In a local system, the HTML would switch between pages by referencing a different HTML page. With our server the actual redirection of the website is handled by the Flask REST API on the server via different GET request handlers.

# **Security.**

## **Reverse Proxy Server.**

For security our project and application needed to implement server security measures as well as website domain security. As mentioned previously within the *Backend & Server* section, our Ubuntu server implements Nginx as a reverse proxy. This proxy server sits on the physical server, acts as a mask and another layer of abstraction from the website domain to the services (Gunicorn in our case) running on the physical server. Nginx lies right behind the physical server’s firewall and is the next gateway needing to be passed through to interact with the Flask REST API.

## **Brute Force Prevention.**

As another form of security we installed and use *Fail2Ban* which is an intrusion prevention software framework. *Fail2Ban* is a commonly used and reliable addition to the security measures as it thwarts brute force attack attempts to breach into the physical server. Any web crawlers and automated bots which ping random IP addresses and attempt to SSH into any server they find, now have their origin IP address banned from connecting again after 3 failed attempts. With *Fail2Ban* we configured a Jail configuration file which references the authentication log of the server and actually handles the IP address banning.

## **Secure Domain.**

Besides on the physical server we installed *Certbot* which is an automated tool which enables HTTPS (HTTP Secured) instead of HTTP for the domain. *Certbot* will configure and automatically renew SSL (Secure Sockets Layer) certification which is required to have a domain registered as HTTPS. This allows our website to ensure better security for the client user as any incoming and outgoing request packets sent are now encrypted.

## **Features Not Needed.**

For the purposes of our project and web application we needed to tackle security in a different sense compared to other projects. Our application currently does not implement any database or storage for data as the focus of our application was *image classification* specifically. We did not implement any form of image data collection as it was not the focus for this application. If we were to implement a page which users could contribute their own pest images then we would most likely use a storage bucket or database system which allows the storage of BLOB images. We would then provide additional security against SQL injections and encrypt all stored data.

# **Testing.**

## **Front End Testing.**

Our website and client side application uses standard JavaScript, HTML, and CSS for the UI and front end logic. For testing in our project we needed a way to verify images being uploaded are valid and safe to send to our server to be processed. The application is simple and testing file uploading mechanics was the only applicable form of testing for our project on the front end side.

The functions necessary to test image uploading were checking filenames, checking file extensions, and checking file size. The check filename testing function verifies if the file being uploaded contains any illegal characters or OS specific unallowed file names. The check file extension function does what the name implies and checks if the file extension is the desired or appropriate type, for our purposes we only accept .jpg or .jpeg files. This is to ensure that a .png file using it’s alpha channel for transparency does not cause a dimensions error with TensorFlow/Keras functions used. Lastly to account for performance and future planned file storage, we implemented a function to test if the file size is at or under 1MB in size.

## **Back End Unit Testing.**

For our Flask REST API there is one POST request handler and 3 GET request handlers corresponding to the 3 pages that are served. As we are able to easily see and test the POST request handler within local development environments and our production environment we omitted this handler from unit testing as it is special and less applicable to unit testing.

On the other hand we implemented 3 important unit tests for the functionality of the GET request handlers. First a function testing for an HTTP 200 “Ok” status is run which verifies that the webpage can be served with no client/server errors. Following this unit test is another which verifies if the content being served from the request is of the right type, which in our case is of the type “text/html” as the GET request handlers serve HTML pages. The third unit testing function implemented actually verifies if the proper data or HTML page is being served to the client by testing for each page, a specific bit of HTML code which only resides on the given page. All of the GET request handler functions are tested with these 3 functions each and all 9 test cases have passed in testing.

## **Server Testing.**

As another important area of our website/application, our server needed special testing. As mentioned in the *Security* section, *Fail2Ban* was implemented and tested to be working by having another group member fail 3 SSH login attempts and their IP address became banned. We of course unbanned their IP address but were able to verify that it works. Our website has been tested to handle multiple users sending multiple requests within the same time and still be able to handle serving them efficiently.

# **Source Control.**

## **GitHub.**

A GitHub repository was used to keep a history of our project, including the frontend and backend code. The code can be accessed through this link: <https://github.com/CWU-BuggyAI/BuggyAI_Source>.

While the repository was helpful in managing code, its best use was in allowing editing from a remote computer and pulling the code onto the server. A user can clone the repository onto their local machine and continue to develop the project without affecting the live website. As improvements accumulate, they can be applied as a hotfix after being reviewed by other teammates. In the case that faulty code is uploaded onto the server, the state can be reverted to a previous version of the project repository.

# **Appendix.**

## **Manual.**

**About BuggyAI.**

BuggyAI is a machine learning image classifier. The application is supposed to classify a given image as one of the following three common pests: aphids, red spider mites, or thrips. The result displayed to the users tells them how confident the program is (in terms of percentage) that the given image is an aphid, or a red spider mite, or a thrips.

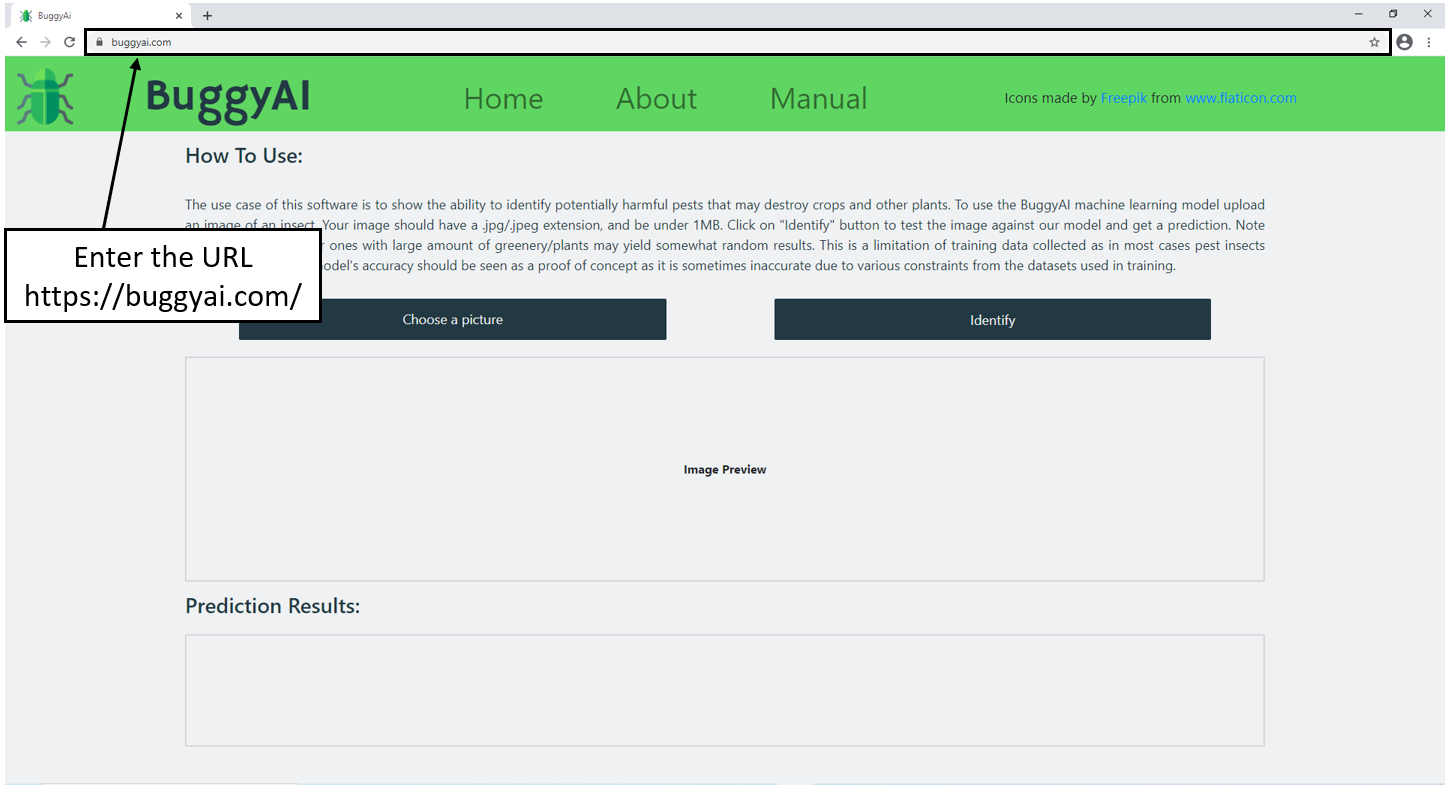
A prediction model was created with Python programming language, using Tensorflow / Keras. After the model was properly trained, it was saved to the backend database. User requests made in the [https://buggyai.com](https://buggyai.com/) will be handled on a Flask application on an Ubuntu 18.04 server which is run in production mode. Gunicorn will then act as a gateway server and handle the execution and entry point of the Flask application. Before all of this, an Nginx server is running as a reverse proxy to filter incoming and outgoing requests being made to the website’s IP address. It is also worth noting that the server has valid SSL certification and only accepts requests from other Https domains.

Before using BuggyAI, make sure your device has a web browser (Google Chrome, Mozilla Firefox, Safari, ...), access to the Internet, and a picture for pest identification readied.

**Getting Started.**

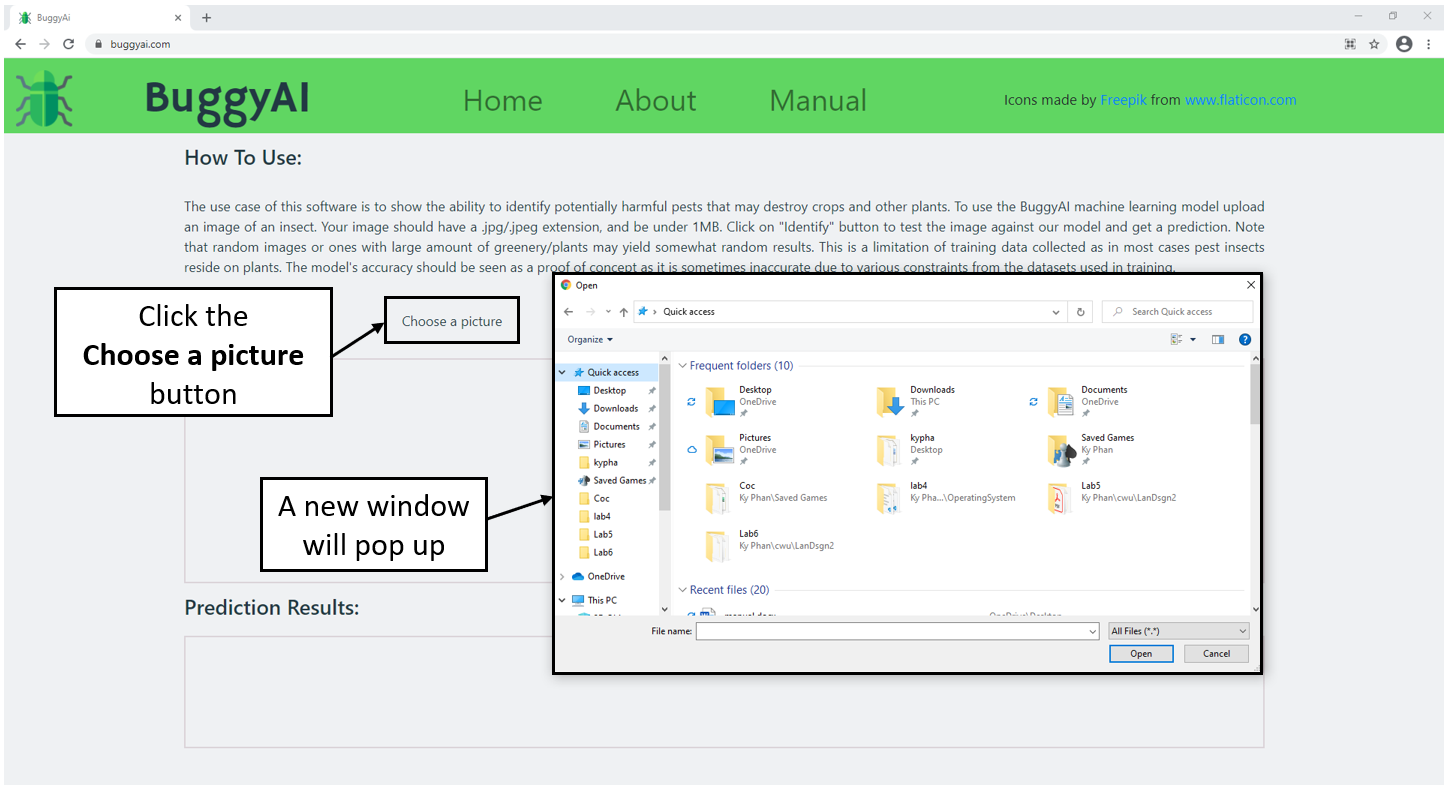
Here is an example on how to use the BuggyAI application on a computer:

1. On your device, open a web browser (Google Chrome, Mozilla Firefox, Safari, ...).
2. Once inside the browser, type in the URL “[https://buggyai.com](https://buggyai.com/)” without quotation marks, and hit **Enter**. The web browser should now display the BuggyAI’s homepage, which looks like this.



Continue instruction on the next page.

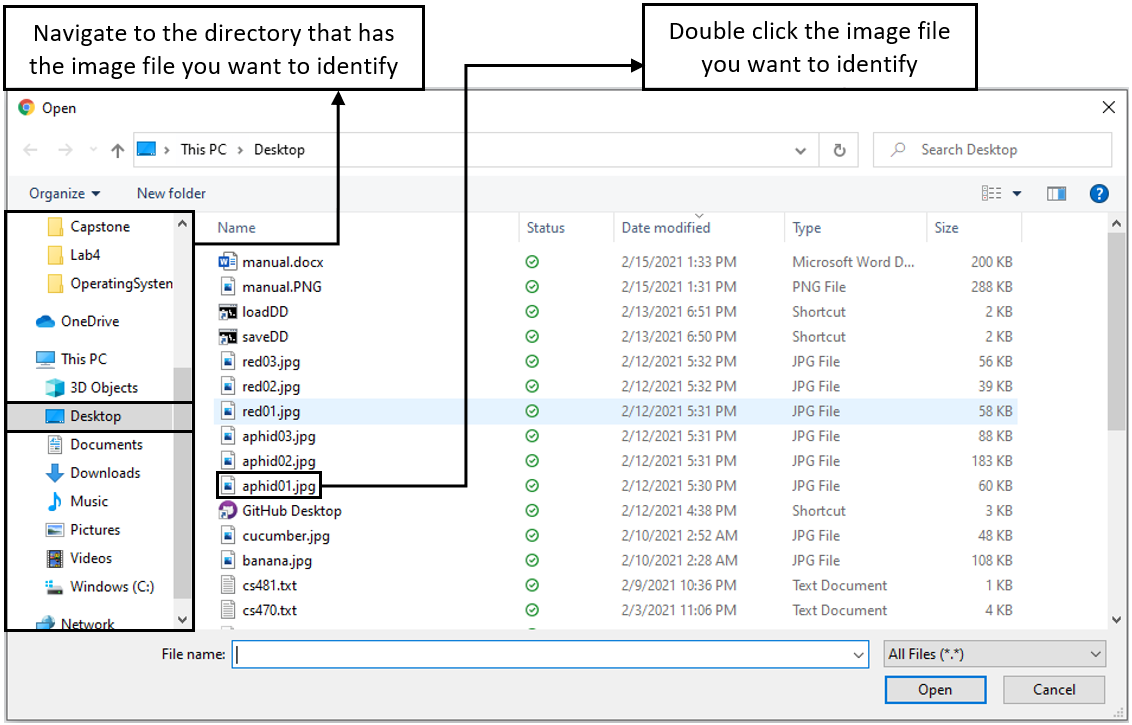
1. Click the **Choose a picture** button. A new window will pop up, and the **Choose a picture** button will turn white, indicating that a picture is needed for the identification.



Continue instruction on the next page.

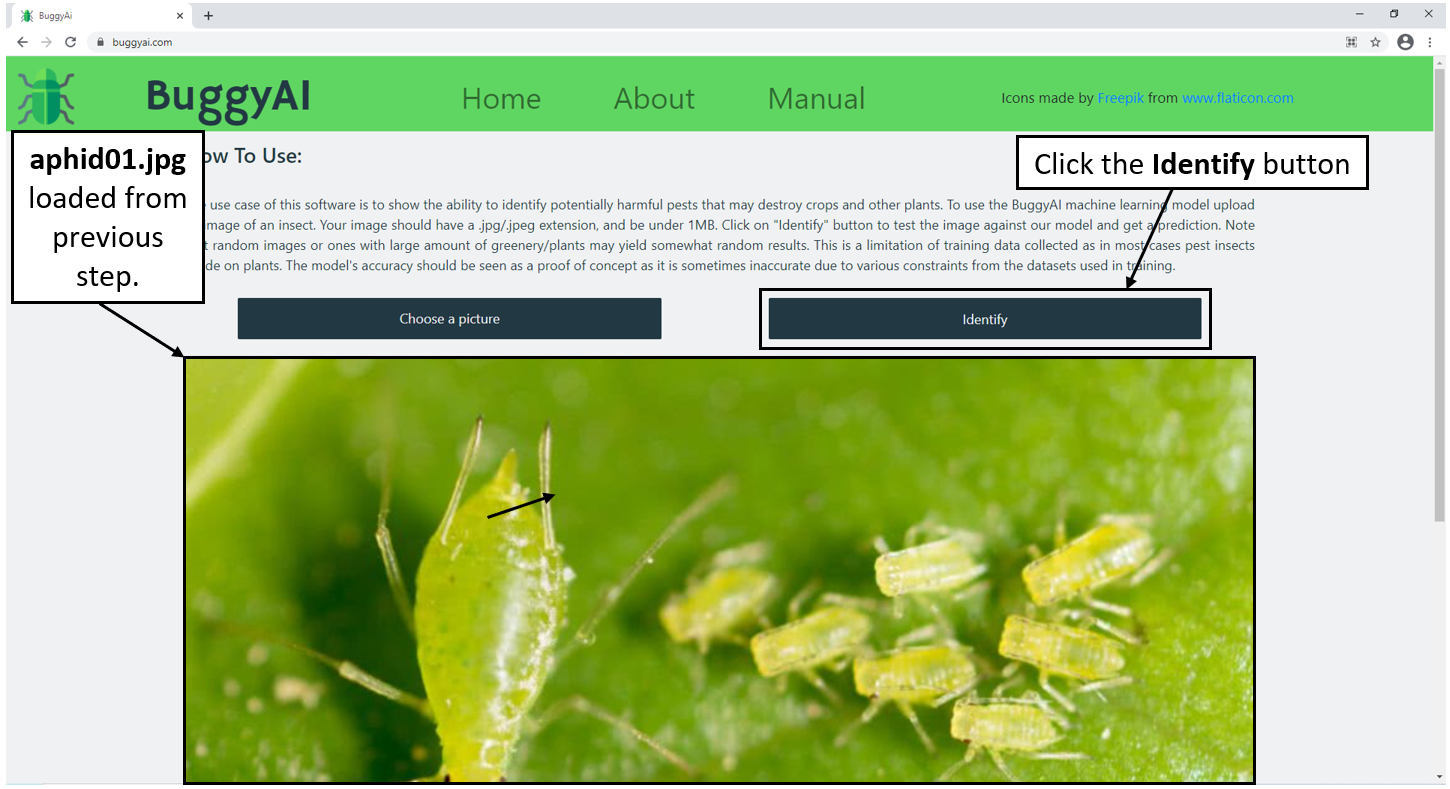
1. From the newly popped up window, navigate to the directory that has the image file you want to identify, double click that image file (for this example, the picture is **aphid01.jpg** inside **Desktop**).

**Important Note**: For now, the application only accepts **.jpg** and **.png** image files.

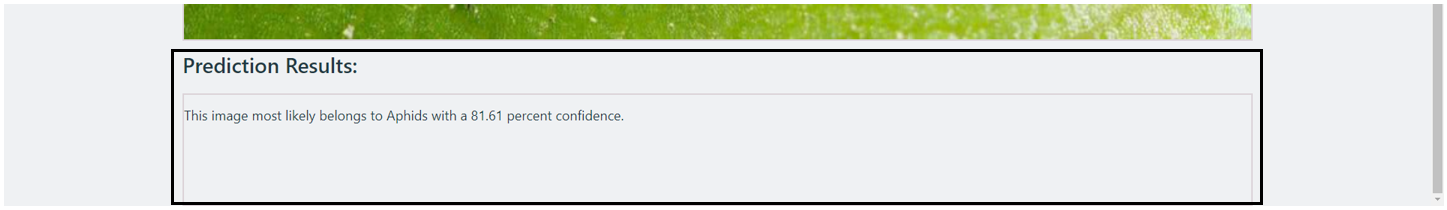


Continue instruction on the next page.

1. If the application can load the selected image file from the previous step 4, it will be shown in the website. Click the **Identify** button to identify.



1. Scroll down to see the result, which will be displayed in the **Prediction Results** area.



In this example, BuggyAI deemed the provided image to be most likely an image of an aphid, with a level of confidence of 81.61 percent.

**The Result.**

**Good results:** If the resulting level of confidence is greater than 66,67 percent (equivalent to the odds of two out of three), then BuggyAI would usually have the correct prediction result.

**Bad results:** If the level of confidence is less than 66,67 percent, then BuggyAI would usually have the wrong prediction result. A low level of confidence means the application was struggling when trying to identify the provided image.

# **References.**

[1] UN. “Have we got enough food to feed the world?”, 2020. [Online]. Available: <https://www.un.org/development/desa/undesavoice/highlights/2020/03/48819.html> (accessed Mar. 10, 2021).

[2] FAO. “New standards to curb the global spread of plant pests and diseases.”, 2019. [Online]. Available: <http://www.fao.org/news/story/en/item/1187738/icode/> (accessed Mar. 10, 2021).

[3] S. Verma. “Understanding Input Output shapes in Convolution Neural Network | Keras.”, 2019. [Online image]. Available: <https://towardsdatascience.com/understanding-input-and-output-shapes-in-convolution-network-keras-f143923d56ca> (accessed Mar. 10, 2021).

[4] Phung, & Rhee. “A High-Accuracy Model Average Ensemble of Convolutional Neural Networks for Classification of Cloud Image Patches on Small Datasets.”, 2019. [Online image]. Available: <https://www.researchgate.net/figure/Schematic-diagram-of-a-basic-convolutional-neural-network-CNN-architecture-26_fig1_336805909> (accessed Mar. 10, 2021).

[5] “What is the number of filter in CNN?” [Online image]. Available: <https://stackoverflow.com/questions/36243536/what-is-the-number-of-filter-in-cnn> (accessed Mar. 10, 2021).

[6] A. Kashyap. “How Convolution Neural Networks interpret images.” 2020. [Online image]. Available: <https://towardsdatascience.com/how-convolution-neural-networks-interpret-images-1f99913070b2> (accessed Mar. 10, 2021).